

```

// -----
// SIMULATEUR DE CHARGE DU RESEAU
// AUTEURS : Beaune Ph., Guschinskaya O., Roelens M.
// DATE : lundi 29 janvier 2007
// Version : 1.0
// -----

// constantes globales

// nombre maximal, fixé a priori, d'appels que l'on pense avoir dans la table
// Communications de la base de données. Valeur à régler selon les cas.
Constante entier MaxAppels = 1000 ;

// nombre maximal, fixé a priori, d'appels en cours, que l'on pense avoir dans
// l'échéancier. Valeur à régler selon les cas.
Constante entier MaxEvénements = 100 ;

// nombre maximal, fixé a priori, de pylones. Valeur à régler selon les cas.
Constante entier MaxPylones = 100 ;

// nombre très grand servant à représenter que 2 pylones ne sont pas
// reliés entre eux
Constante entier Infini = 99999 ;

// structures de données

// la structure Statistiques permettra de mémoriser pour chaque pylone les
// informations nécessaires au calcul des indicateurs de performances, à
// savoir le nombre max et le nombre moyen de communications de chaque pylone
structure Statistiques :
// nombre maximum de communications pour 1 pylone :
NbMaxComm : entier ;
// nombre courant, à tout instant, de communications pour 1 pylone :
NbCourantComm : entier ;
// permettra de calculer le nombre moyen de communications pour 1 pylone
// en divisant cette grandeur par la longueur de l'intervalle de temps
// considéré :
PourNbCommMoyen : entier ;
finstructure ;

// la structure Pylone contiendra les informations connues au départ : les
// coordonnées et la portée de chaque pylone. Ces informations permettront
// le calcul de la matrice des prédécesseurs dans les plus courts chemins.
// Cette structure contiendra aussi de quoi mémoriser les indicateurs de
// performance de chaque pylone. L'ensemble des pylones sera regroupé dans
// un tableau, ainsi chaque pylone pourra être désigné par son numéro, à
// savoir l'index dans ce tableau.

structure Pylone :
// coordonnées X et Y du pylone :
X , Y : entier ;
// portée du pylone
Portée : entier ;
// indicateurs de performances du pylone :
IndicPerf : Statistiques ;
finstructure ;

// dans cette structure on mémorisera tous les pylones à partir d'informations
// fournies dans une feuille du tableur. Comme on ne sait pas a priori combien
// il y aura de pylones, on disposera d'un entier qui indiquera le rang du
// dernier pylone initialisé dans la liste.
structure ListePylones :
// index du dernier pylone initialisé :
NbPylones : entier ;
// liste de tous les appels triés :
Liste : Pylone[0..MaxPylones] ;
finstructure ;

```

```

// la structure Appel mémorisera les seules informations utiles pour le
// simulateur, parmi toutes les informations disponibles pour chaque
// communication.
structure Appel :
    // date de début et de fin d'une communication :
    Début, Fin : entier ;
    // pylones entrant et sortant de la communication :
    NumPyloneEntrant, NumPyloneSortant : entier ;
finstructure ;

// dans cette structure on mémorisera tous les appels extraits de la table
// Communications de la base de données. Dans cette structure, les appels
// seront triés par date de début de communication. Comme on va parcourir
// un à un tous ces appels, on aura aussi besoin d'un index du prochain appel
// à traiter. On saura que tous les appels ont été traités lorsque cet
// index vaudra -1.
structure ListeAppels :
    // index du prochain appel à traiter :
    ProchainAppel : entier ;
    // liste de tous les appels triés :
    Liste : Appel[0..MaxAppels] ;
finstructure ;

// Un événement est uniquement une fin de communication. La structure Evénement
// permettra de mémoriser la date de cette fin de communication, ainsi que les
// 2 pylones (entrant et sortant) concernés par cette communication.
structure Evénement :
    // date de l'événement, à savoir date de fin de la communication :
    Date : entier ;
    // numéros des 2 pylones concernés :
    NumPyloneEntrant, NumPyloneSortant : entier ;
finstructure ;

// la fin de chaque appel en cours à tout instant sera mémorisée dans un
// échéancier, trié par date décroissante, c'est à dire que l'événement de
// date minimum se trouvera en tête de tableau, du côté du plus grand index.
// Cela permettra de dépiler le prochain événement sans avoir à décaler tous
// les autres événements. Un index indique quel est le prochain événement à
// traiter. On saura que l'échéancier est vide lorsque cet index vaudra -1.
structure Echéancier :
    // index du prochain événement :
    Index : entier ;
    // liste de les événements :
    Liste : Evénement[0..MaxEvénements] ;
finstructure ;

```

```
// programme principal, qui vous sera fourni, et qui sera associé
// à un bouton d'une feuille du tableur.
```

```
Programme LancerSimulation()
```

```
{
    // liste de tous les pylones :
    MesPylones : ListePylones ;

    // matrice des prédécesseurs dans les plus courts chemins : chaque pylone
    // est repéré par son numéro, c'est à dire son index dans MesPylones
    MesPrédécesseurs : entier[0..MaxPylones , 0..MaxPylones] ;

    // liste des tous les appels triés :
    MesAppels : ListeAppels ;

    // procédure qui initialise le tableau de mes pylones
    ChargerPylones(MesPylones) ;

    // procédure qui calcule, à partir des coordonnées et des portées
    // des pylones, la matrice des prédécesseurs dans les plus courts
    // chemins entre toutes paires de pylones
    CalculerPrédécesseurs(MesPylones , MesPrédécesseurs) ;

    // procédure qui initialise la liste de mes appels en les triant
    // par date de début de communication
    ChargerAppels(MesAppels) ;

    // procédure du simulateur proprement dit : calcule les indicateurs
    // de performances de chaque pylone, à partir de la matrice des
    // prédécesseurs et de la liste des appels
    Simuler(MesPylones , MesPrédécesseurs , MesAppels) ;

    // procédure d'affichage des indicateurs de performances de tous
    // les pylones
    AfficherIndicateurs(MesPylones) ;
};
```

```
// procédure qui initialise le tableau de mes pylones à partir d'une
// feuille du tableur. Ci-dessous cette feuille du tableur et le mode
// d'accès aux informations qu'elle contient est nommée « source ».
```

```
Procédure ChargerPylones(réf P : ListePylones) :
```

```
{
    P.NbPylones = -1 ;

    TantQue (source contient un pylone)
        P.NbPylones = P.NbPylones + 1 ;
        Si P.NbPylones > MaxPylones alors
            StopProgramme(« Trop de Pylones ! »);
        FinSi ;

        P.Liste[P.NbPylones].X = (x in source) ;
        P.Liste[P.NbPylones].Y = (y in source) ;
        P.Liste[P.NbPylones].Portée = (portée in source) ;
    FinTantQue ;
}
```

```
// procédure qui initialise le tableau de mes appels à partir d'une
// feuille du tableur. Ci-dessous cette feuille du tableur et le mode
// d'accès aux informations qu'elle contient est nommée « source ». Cette
// procédure doit aussi trier les appels par dates de début de
// communication.
```

Procédure ChargerAppels(réf A : ListeAppels) :

```
{
  i : entier ;
  A.ProchainAppel = -1 ;
  TantQue (source contient un appel)
  A.ProchainAppel = A.ProchainAppel + 1 ;
  Si A.ProchainAppel > MaxAppels alors
    StopProgramme(« Trop d'appels ! »);
  FinSi ;

  // recherche du bon emplacement i pour insérer cet appel
  i = A.ProchainAppel ;
  TantQue ( ( i > 0 ) et (A.Liste[i-1].Début < (début in source)) )
    i = i - 1 ;
  A.Liste[i+1] = A.Liste[i] ;
  FinTantQue ;

  A.Liste[i].Début = (début in source) ;
  A.Liste[i].Fin = (fin in source) ;
  A.Liste[i].NumPyloneEntrant = (numéro pylone entrant in source) ;
  A.Liste[i].NumPyloneSortant = (numéro pylone sortant in source) ;
  FinTantQue ;
}
```

// procédure de calcul de la distance euclidienne entre 2 points

réel Fonction Distance(X1 , Y1 , X2 , Y2 : entier) :

```
{
  Retour(RacineCarrée(((X1 - X2)**2) + ((Y1 - Y2)**2))) ;
}
```

// procédure qui calcule la matrice d'adjacence des pylones à partir
// des coordonnées X et Y de chaque pylone. Cette procédure est ensuite
// appelée par CalculerPrédécesseurs. Rappel : la distance entre les
// pylones n'importe pas : on notera 1 si deux pylones sont adjacents, infini
// sinon. On considère que tout pylone est relié à lui-même.
// A noter : étant donné le choix « deux pylones sont adjacents si leurs
// portées s'intersectent », la matrice est symétrique : on n'en parcourt
// donc que la moitié.

Procédure CalculerAdjacence(réf P : ListePylones ,
réf M : entier[0..MaxPylones , 0..MaxPylones]) :

```
{
  i , j : entier ;
  d : réel ;
  Pour i = 0 à P.NbPylones ParPasDe 1
    M[i,i] = 1 ;
    Pour j = i+1 à P.NbPylones ParPasDe 1
      d = Distance( P.Liste[i].X , P.Liste[i].Y , P.Liste[j].X,P.Liste[j].Y ) ;
      Si (d <= (P.Liste[i].Portée + P.Liste[j].Portée) )
        alors
          M[i,j] = 1 ;
          M[j,i] = 1 ;
        sinon
          M[i,j] = Infini ;
          M[j,i] = Infini ;
        FinSi ;
      FinPour ;
    FinPour ;
  FinPour ;
}
```

```
// procédure qui calcule la matrice des prédécesseurs à partir des
// coordonnées et des portées des pylones. Méthode de Dantzig.
```

```
Procédure CalculerPrédécesseurs( réf P : ListePylones ,
                                réf M : entier[0..MaxPylones , 0..MaxPylones] ) :
{
  MatAdj : entier[0..MaxPylones , 0..MaxPylones] ;
  Dist : entier[0..MaxPylones , 0..MaxPylones] ;
  i , j , k : entier ;

  CalculerAdjacence(P , MatAdj) ;

  // on initialise la matrice des prédécesseurs avec les arcs du graphe
  Pour i=0 à P.NbPylones ParPasDe 1
    Pour j=0 à P.NbPylones ParPasDe 1
      si (MatAdj[i,j]=1)
        alors
          M[i,j] = i ;
        sinon
          M[i,j] = 0 ;
        FinSi ;
      FinPour ;
    FinPour ;

  // on initialise la matrice Dist des plus courts chemins avec 0
  // sur la diagonale, et Infini partout ailleurs.
  Pour i=0 à P.NbPylones ParPasDe 1
    Pour j=0 à P.NbPylones ParPasDe 1
      si (i=j)
        alors
          Dist[i,i] = 0 ;
        sinon
          Dist[i,j] = Infini ;
        FinSi ;
      FinPour ;
    FinPour ;

  // on fait entrer les nouveaux sommets un par un
  Pour k=1 à P.NbPylones ParPasDe 1

    Pour i=0 à k-1 ParPasDe 1
      Pour j=0 à k-1 ParPasDe 1

        // on regarde les chemins de i à k terminant par l'arc j-k
        Si (Dist[i,k] > Dist[i,j] + MatAdj[j,k]) Alors
          M[i,k] = j ;
          Dist[i,k] = Dist[i,j] + MatAdj[j,k] ;
        FinSi ;

        // on regarde les chemins de k à i commençant par l'arc k-j
        Si (Dist[k,i] > MatAdj[k,j] + Dist[j,i]) Alors
          M[k,i] = M[j,i] ;
          Dist[k,i] = MatAdj[k,j] + Dist[j,i] ;
        FinSi ;
      FinPour ;
    FinPour ;

  // on regarde les chemins de i à j passant par k
  Pour i=0 à k-1 ParPasDe 1
    Pour j=0 à k-1 ParPasDe 1
      Si (Dist[i,j] > Dist[i,k] + Dist[k,j]) Alors
        M[i,j] = M[k,j] ;
        Dist[i,j] = Dist[i,k] + Dist[k,j] ;
      FinSi ;
    FinPour ;
  FinPour ;
}

```

```
// mise à jour de l'indicateur qui permettra de calculer le taux moyen
// d'occupation de chaque pylone
```

```
Procédure MettreAJourPourNbCommMoyen( réf P ListePylones,  
                                     TpsEvtPrécédent : entier,  
                                     TpsEvtCourant : entier)  
  
{  
    i : entier ;  
  
    Pour i = 0 à P.NbPylones ParPasDe 1  
        P.ListePylones[i].IndicPerf.PourNbCommMoyen =  
            P.ListePylones[i].IndicPerf.PourNbCommMoyen +  
            P.ListePylones[i].IndicPerf.NbCourantComm * (TpsEvtCourant - TpsEvtPrécédent) ;  
    FinPour ;  
}
```

```
// mise à jour le nombre courant et le nombre max de comm de chaque pylone  
// intermédiaire entre un pylone entrant et un pylone sortant
```

```
Procédure MettreAJourNbCourantEtMax ( Incrément : entier,  
                                     réf P : ListePylones,  
                                     réf M : entier[0..MaxPylones , 0..MaxPylones],  
                                     NumPyloneEntrant : entier,  
                                     NumPyloneSortant : entier)  
  
{  
    i : entier;  
  
    i = NumPyloneSortant ;  
    TantQue (i /= NumPyloneEntrant)  
        P.Liste[i].IndicPerf.NbCourantComm = P.Liste[i].IndicPerf.NbCourantComm + Incrément ;  
        Si (P.Liste[i].IndicPerf.NbCourantComm > P.Liste[i].IndicPerf.NbMaxComm) Alors  
            P.Liste[i].IndicPerf.NbMaxComm = P.Liste[i].IndicPerf.NbCourantComm ;  
        FinSi ;  
        i = M[NumPyloneEntrant,i] ;  
    FinTantQue ;  
  
    // Cas du pylone NumPyloneEntrant  
    P.Liste[i].IndicPerf.NbCourant = P.Liste[i].IndicPerf.NbCourant + Incrément ;  
    Si (P.Liste[i].IndicPerf.NbCourantComm > P.Liste[i].IndicPerf.NbMaxComm) Alors  
        P.Liste[i].IndicPerf.NbMaxComm = P.Liste[i].IndicPerf.NbCourantComm ;  
    FinSi ;  
}
```

```
// insertion d'un nouvel événement dans l'échéancier
```

```
Procédure InsérerEchéancierFinAppel(réf E : Echancier, A : Appel)  
{  
    i : entier ;  
  
    E.Index = E.Index + 1 ;  
    Si E.Index > MaxEvénements alors  
        StopProgramme(« Trop d'événements ! »);  
    FinSi ;  
  
    // recherche du bon emplacement i pour insérer cette fin d'appel  
    i = E.Index ;  
    TantQue ( i > 0) et (E.Liste[i].Date < A.Fin) )  
        i = i - 1 ;  
        E.Liste[i+1] = E.Liste[i] ;  
    FinTantQue ;  
  
    E.Liste[i].Date = A.Fin ;  
    E.Liste[i].NumPyloneEntrant = A.NumPyloneEntrant ;  
    E.Liste[i].NumPyloneSortant = A.NumPyloneSortant ;  
}
```

```
// affichage des indicateurs de tous les pylones
```

```
Procédure AfficherIndicateurs(réf P : ListePylones)  
{  
    i : entier ;  
  
    Pour i = 0 à P.NbPylones ParPasDe 1  
        (moyenne de i in feuille tableur) = P.Liste[i].IndicPerf.PourNbCommMoyen ;  
        (max de i in feuille tableur) = P.Liste[i].IndicPerf.NbMaxComm ;  
    FinPour ;  
}
```

```
// procédure qui calcule les indicateurs de chaque pylone à partir de
// la liste de tous les appels et de la matrice des prédécesseurs.
```

```
Procédure Simuler( réf P : ListePylones ,
                  réf M : entier[0..MaxPylones , 0..MaxPylones] ,
                  réf A : ListeAppels) :
{
  i : entier ;
  E : Echancier ;
  DébutSimulation, TpsEvtPrécédent, TpsEvtCourant : entier ;

  E.Index = -1 ;
  DébutSimulation = A.Liste[A.ProchainAppel].Début ;
  TpsEvtCourant = DébutSimulation ;

  // mise à 0 des indicateurs des pylones
  Pour i = 0 à P.NbPylones ParPasDe 1
    P.Liste[i].IndicPerf.NbMaxComm = 0 ;
    P.Liste[i].IndicPerf.NbCourantComm = 0 ;
    P.Liste[i].IndicPerf.PourNbCommMoyen = 0 ;
  FinPour ;

  TantQue ( (E.Index > -1) ou (A.ProchainAppel > -1) )
    // L'échéancier n'est pas vide ou il reste des appels à traiter

    Si ((E.Index = -1) ou
        ((A.ProchainAppel > -1) et (A.Liste[A.ProchainAppel] < E.Liste[E.Index])))
      Alors
        // l'échéancier est vide ou le prochain appel est plus proche que la
        // prochaine fin de communication : on traite un début d'appel
        TpsEvtPrécédent = TpsEvtCourant ;
        TpsEvtCourant = A.Liste[A.ProchainAppel].Début ;

        MettreAJourPourNbCommMoyen(P, TpsEvtPrécédent, TpsEvtCourant) ;

        MettreAJourNbCourantEtMax( +1 , P , M ,
                                  A.Liste[A.ProchainAppel].NumPyloneEntrant ,
                                  A.Liste[A.ProchainAppel].NumPyloneSortant) ;

        InsérerEchancierFinAppel(E , A.Liste[A.ProchainAppel]) ;

        // on passe au prochain appel
        A.ProchainAppel = A.ProchainAppel - 1 ;

      Sinon
        // Il n'y a plus d'appels ou le prochain appel est moins proche que (ou
        // égale à) la prochaine fin de communication : on traite une fin d'appel
        TpsEvtPrécédent = TpsEvtCourant ;
        TpsEvtCourant = E.Liste[E.Index].Date ;

        MettreAJourPourNbCommMoyen(P, TpsEvtPrécédent, TpsEvtCourant) ;

        MettreAJourNbCourantEtMax( -1 , P , M ,
                                  E.Liste[E.Index].NumPyloneEntrant ,
                                  E.Liste[E.Index].NumPyloneSortant) ;

        // on passe au prochain événement
        E.Index = E.Index - 1;

      FinSi ;

    FinTantQue ;

  // calcul des nombres moyens de comm pour chaque pylone
  Pour i = 0 à P.NbPylones ParPasDe 1
    P.Liste[i].IndicPerf.PourNbCommMoyen =
      P.Liste[i].IndicPerf.PourNbCommMoyen / (TpsEvtCourant - DébutSimulation) ;
  FinPour ;
}
```